The "PAQJP_6_Smart" compression project:

Overview

The project combines several lossless compression strategies:

- **Markers**: Indicate the compression method used (`01` for PAQJP_6 with transformations, `02` for zlib without transformation, `03` for zlib with transformations, `04` for Huffman coding).

- **Transformations (1–255)**: Reversible data transformations applied before compression to enhance compressibility.

- **Dictionary**: A set of preloaded text files used for hash-based verification.

- **Huffman Coding**: Used for small files (<1024 bytes) under marker `04` or `01`.

All components are designed to be **100% lossless**, ensuring exact recovery of the original data.

Compression Markers

Marker `01` (PAQJP_6 Compressor)

- **Description**: This method applies one of the transformations (1–255) to the input data, followed by either PAQ, zlib, or Huffman compression, depending on file size and mode (fast or slow). The transformation used is indicated by a marker byte (1–255) in the output.

- **Process**:

  1. A transformation (1–255) is applied to the input data to make it more compressible.

  2. The transformed data is compressed using:

     - **PAQ**: For larger files, using the `paq_compress` function (PAQ9a, a lossless arithmetic coding algorithm).

     - **zlib**: For medium-sized files, using `compress_data_zlib` (a standard lossless compression algorithm).

     - **Huffman**: For small files (<1024 bytes), using `compress_data_huffman` (see Huffman section below).

  3. The output format is: `[01][transform_marker][compressed_data]`, where `transform_marker` (1–255) indicates the transformation used.

- **Lossless Property**:

  - All transformations are reversible (see Transformations section).

  - PAQ, zlib, and Huffman are lossless compression algorithms.

  - During decompression (`decompress_with_best_method`), the transform marker identifies the correct reverse transformation, ensuring exact data recovery.

- **Use Case**: Versatile for various file types, with prioritized transformations (7–9, 10–255 in slow mode) for JPEG and text files.

Marker `02` (zlib No Transform)

- **Description**: This method applies zlib compression directly to the input data without any transformation.

- **Process**:

  1. The input data is compressed using `compress_data_zlib` (zlib's lossless compression).

  2. The output format is: `[02][compressed_data]`.

- **Lossless Property**:

  - zlib is inherently lossless, preserving all original data bits.

  - Decompression (`decompress_data_zlib`) restores the exact input data.

- **Use Case**: Suitable for files where transformations do not improve compression, providing a simple and fast compression option.

Marker `03` (zlib with Transform)

- **Description**: This method applies one of the transformations (1–255) to the input data, followed by zlib compression. The transformation is indicated by a marker byte (1–255).

- **Process**:

  1. A transformation (1–255) is applied to enhance compressibility.

  2. The transformed data is compressed using `compress_data_zlib`.

  3. The output format is: `[03][transform_marker][compressed_data]`.

- **Lossless Property**:

  - Transformations are reversible (see Transformations section).

  - zlib compression is lossless.

  - Decompression (`decompress_with_best_method`) uses the transform marker to apply the correct reverse transformation, ensuring exact recovery.

- **Use Case**: Effective for files where transformations improve zlib's compression ratio, especially for JPEG and text files.

Marker `04` (Huffman Coding)

- **Description**: This method applies Huffman coding to small files (<1024 bytes), typically used within marker `01` but can be considered a standalone marker for small data.

- **Process**:

  1. The input data is converted to a binary string (e.g., `bin(int(binascii.hexlify(data), 16))[2:]`).

  2. Huffman codes are generated based on bit frequencies (`calculate_frequencies`, `build_huffman_tree`, `generate_huffman_codes`).

  3. The binary string is encoded into a compressed binary string (`compress_data_huffman`).

  4. The compressed string is converted to bytes and prefixed with marker `04` (or included under `01` with transform marker `4`).

- **Lossless Property**:

  - Huffman coding is lossless, as it assigns unique, variable-length codes to each symbol (bit) based on frequency, allowing exact reconstruction.

  - Decompression (`decompress_data_huffman`) rebuilds the Huffman tree and decodes the binary string to the original data.

- **Use Case**: Optimized for small files where PAQ or zlib may be less efficient.

Transformations (1–255)

Transformations are reversible operations applied to the input data before compression to improve compressibility. They are used in markers `01` and `03`. Each transformation has a corresponding reverse transformation, ensuring **100% losslessness**. Below is an overview of the transformations:

- **Transformation 1 (transform_04/reverse_transform_04)**:

  - **Operation**: Subtracts position-based values modulo 256 (`(x - i % 256) % 256`).

  - **Reverse**: Adds the same values modulo 256 (`(x + i % 256) % 256`).

  - **Lossless**: The operation is bijective, as modular arithmetic ensures unique mappings.

- **Transformation 2 (transform_01/reverse_transform_01)**:

  - **Operation**: XORs every third byte with prime numbers (`transform_with_prime_xor_every_3_bytes`).

  - **Reverse**: Applies the same XOR operation, as XOR is symmetric (`x ^ k ^ k = x`).

  - **Lossless**: Symmetric XOR ensures exact reversal.

- **Transformation 3 (transform_03/reverse_transform_03)**:

  - **Operation**: XORs data in chunks of 4 bytes with `0xFF` (`transform_with_pattern_chunk`).

  - **Reverse**: Applies the same XOR, as it is symmetric.

  - **Lossless**: Symmetric operation guarantees no data loss.

- **Transformation 5 (transform_05/reverse_transform_05)**:

  - **Operation**: Performs a left bit shift with rotation (`(x << shift) | (x >> (8 - shift)) & 0xFF`).

  - **Reverse**: Performs a right bit shift with rotation.

  - **Lossless**: Bit rotations preserve all bits, ensuring reversibility.

- **Transformation 6 (transform_06/reverse_transform_06)**:

  - **Operation**: Uses a seeded substitution table to remap byte values.

  - **Reverse**: Uses the inverse substitution table, generated from the same seed.

  - **Lossless**: The substitution is a permutation, ensuring a one-to-one mapping.

- **Transformation 7 (transform_07/reverse_transform_07)**:

  - **Operation**: XORs with a size-based byte (`len(data) % 256`) and pi digits, cycled based on data size.

  - **Reverse**: Applies the same XOR operations in reverse order, adjusting pi digits accordingly.

  - **Lossless**: Symmetric XOR and consistent pi digit usage ensure reversibility.

- **Transformation 8 (transform_08/reverse_transform_08)**:

  - **Operation**: XORs with the nearest prime to `len(data) % 256` and pi digits.

  - **Reverse**: Reverses the XOR operations, using the same prime and pi digits.

  - **Lossless**: Symmetric and deterministic operations ensure no data loss.

- **Transformation 9 (transform_09/reverse_transform_09)**:

  - **Operation**: XORs with a prime, a seed value from a table, and pi digits.

  - **Reverse**: Reverses the XOR operations with the same values.

  - **Lossless**: All operations are symmetric and reversible.

- **Transformation 10 (transform_10/reverse_transform_10)**:

  - **Operation**: XORs with a computed value `n` based on the count of "X1" sequences in the data.

  - **Reverse**: XORs with the stored `n` (included in the output as a prefix byte).

  - **Lossless**: The value `n` is stored, ensuring exact reversal.

- **Transformation 11 (transform_11/reverse_transform_11)**:

  - **Operation**: Adds an optimal `y` value (1–255) modulo 256 to minimize compressed size.

  - **Reverse**: Subtracts the stored `y` value (prefixed in the output).

  - **Lossless**: Modular addition/subtraction is bijective.

- **Transformations 12–255 (generate_transform_method)**:

  - **Operation**: XORs with a value derived from data size and position (`(size_mod + i % 256) % 256`).

  - **Reverse**: Applies the same XOR operation, as it is symmetric.

  - **Lossless**: The XOR operation is reversible and deterministic.

- **General Lossless Property**:

  - Each transformation is bijective (one-to-one and onto), ensuring that applying the transformation and its reverse restores the original data exactly.

  - Transformations are paired with their reverses in the code (`reverse_transform_XX`), guaranteeing losslessness.

  - The transform marker (1–255) ensures the correct reverse transformation is applied during decompression.

Dictionary Functionality

- **Description**: The system uses a set of predefined dictionary files (listed in `DICTIONARY_FILES`) to verify the integrity of compressed data via SHA-256 hash matching.

- **Process**:

  1. For input data, a SHA-256 hash is computed (`compute_sha256`).

  2. The hash is searched in dictionary files (`find_hash_in_dictionaries`) to check if the data is known.

  3. If a match is found, it's logged, but compression proceeds normally (no data substitution occurs).

  4. For `.paq` files (e.g., `words.txt.paq`), an 8-byte SHA-256 hash may be returned instead of compressed data if it's smaller.

- **Lossless Property**:

  - The dictionary is used only for verification, not for altering data.

  - The SHA-256 hash ensures that decompressed data matches the original, but it's not used in markers `01`, `02`, `03`, or `04` directly (it's primarily for marker `00`, which is excluded here).

  - For `.paq` files, the hash is a direct representation of the data, and decompression checks the hash, ensuring losslessness.

- **Use Case**: Enhances security by verifying data integrity against known dictionaries, particularly for specific file types.

Huffman Compression

- **Description**: Huffman coding is used for small files (<1024 bytes) under marker `01` (with transform marker `4`) or as a standalone option (marker `04` in some contexts).

- **Process**:

  1. **Compression** (`compress_data_huffman`):

     - Convert input data to a binary string (bit sequence).

     - Compute frequencies of bits (`0` and `1`) using `calculate_frequencies`.

     - Build a Huffman tree (`build_huffman_tree`) based on bit frequencies.

     - Generate variable-length codes (`generate_huffman_codes`).

     - Encode the binary string into a compressed binary string using Huffman codes.

     - Convert the compressed string to bytes for output.

  2. **Decompression** (`decompress_data_huffman`):

     - Rebuild the Huffman tree from the compressed data's bit frequencies.

     - Decode the binary string back to the original bit sequence using the Huffman codes.

     - Convert the bit sequence to bytes.

- **Lossless Property**:

  - Huffman coding assigns unique codes to each symbol (bit), ensuring no information loss.

  - The decompression process reconstructs the exact original bit sequence.

  - The code handles edge cases (e.g., empty inputs) by returning empty results, avoiding data corruption.

- **Use Case**: Efficient for small files where PAQ or zlib overhead is significant, providing compact compression.

Lossless Guarantee

- **Markers `01`, `02`, `03`, `04`**:

  - Marker `01` ensures losslessness through reversible transformations and lossless compression (PAQ, zlib, or Huffman).

  - Marker `02` uses zlib directly, which is lossless.

  - Marker `03` combines reversible transformations with lossless zlib compression.

  - Marker `04` uses Huffman coding, which is inherently lossless.

- **Transformations 1–255**:

  - Each transformation is mathematically reversible, using operations like XOR, modular arithmetic, bit shifts, or permutations, all of which are bijective.

  - The transform marker ensures the correct reverse transformation is applied.

- **Dictionary**:

  - Used for verification, not data alteration, preserving losslessness.

  - For `.paq` files, the hash-based approach ensures exact recovery.

- **Huffman**:

  - Provides lossless compression for small files, with exact reconstruction of the original data.

- **Error Handling**:

  - The code includes robust checks for empty inputs, invalid markers, and compression/decompression failures, ensuring no data corruption.

  - Logging of zero-byte counts aids debugging but does not affect losslessness.

Summary

The "PAQJP_6_Smart" project’s markers `01`, `02`, `03`, and `04` are **100% lossless**, leveraging reversible transformations and lossless compression algorithms (PAQ, zlib, Huffman). Transformations 1–255 are reversible, ensuring no data loss when paired with the correct reverse transformation. The dictionary functionality enhances verification without affecting data, and Huffman coding provides efficient, lossless compression for small files. The system is robust, with comprehensive error handling to maintain data integrity across all operations.